BRIDGING C++ AND PYTHON WITH NANOBIND

CPP USERGROUP VIENNA

Maximilian Kleinert

3.6.2025

MOTIVATION

- **Performance**: C++ offers high performance for compute-intensive tasks
- **Productivity**: Offer a Python interface for C++ libraries to leverage Python's ease of use
- **Reuse**: Leverage existing C++ libraries in Python projects without rewriting code
- **Prototyping**: Rapidly prototype in Python, optimize bottlenecks in C++

HISTORIC OVERVIEW

- **Boost.Python** (2002, Dave Abrahams)
    - Early, powerful, but heavy-weight and complex
    - Part of the Boost C++ Libraries
    - Required linking against Boost and Python

- **pybind11** (2016, Wenzel Jakob)
    - Inspired by Boost.Python, but header-only and lightweight
    - Modern C++11/14 support
    - Widely adopted in open source and industry

- **nanobind** (2022, Wenzel Jakob)
  - Successor to pybind11, designed for Python 3.8+
  - Smaller, faster, and more memory efficient
  - Focuses on minimalism and performance

SIMPLE NANOBIND BINDING EXAMPLE

```cpp
// example.cpp
#include <nanobind/nanobind.h>
namespace nb = nanobind;

int add(int a, int b) {
    return a + b;
}

NB_MODULE(my_ext, m) {
    m.def("add", &add, "Add two numbers");
}
```

USING FROM PYTHON

```python
import my_ext
print(my_ext.add(2, 3))  # Output: 5
```

EXCHANGING INFORMATION: ARGUMENTS AND RETURN VALUES

- nanobind automatically converts between C++ and Python types for:
  - Fundamental types: `int`, `float`, `bool`, etc.
  - STL types: `std::string`, `std::vector`, `std::map`, etc.
    (requires extra nanobind headers)
- Custom classes when you create bindings

EXAMPLE: PASSING AND RETURNING STRINGS

```cpp
#include <nanobind/nanobind.h>
#include <nanobind/stl/string.h>
namespace nb = nanobind;

std::string greet(const std::string &name) {
    return "Hello, " + name + "!";
}

NB_MODULE(my_ext, m) {
    m.def("greet", &greet, "Greet someone by name");
}
```

```python
import my_ext
print(my_ext.greet("World"))  # Output: Hello, World!
```

EXAMPLE: WORKING WITH LISTS

```cpp
#include <nanobind/nanobind.h>
#include <nanobind/stl/vector.h>  // Required for std::vector sup
#include <vector>
namespace nb = nanobind;

int sum_vector(const std::vector<int> &v) {
    int sum = 0;
    for (int x : v) sum += x;
    return sum;
}

NB_MODULE(my_ext, m) {
    m.def("sum_vector", &sum_vector, "Sum a list of integers");
}
```

```python
import my_ext
print(my_ext.sum_vector([1, 2, 3]))  # Output: 6
```

DOWNSIDES: WORKING WITH LISTS AND COPIES

- When passing a Python `list` to a C++ function expecting a `std::vector`, **nanobind creates a copy** of the data
- This can be inefficient for large arrays or performance-critical code
- The copy happens both when passing data from Python to C++ and when returning a `std::vector` to Python

BETTER OPTIONS: EIGEN AND NUMPY ARRAYS

- For numerical data and large arrays, prefer using **Eigen** or **NumPy** arrays
- nanobind provides **type casters** for Eigen and NumPy, enabling **zero-copy** data exchange
- This allows C++ functions to operate directly on Python memory buffers, avoiding unnecessary copies

EXAMPLE: USING NUMPY ARRAYS (ZERO-COPY)

```cpp
#include <nanobind/nanobind.h>
#include <nanobind/ndarray.h>
namespace nb = nanobind;

float sum_array(nb::ndarray<nb::numpy, float, nb::c_contig> arr)
    float sum = 0;
    for (ssize_t i = 0; i < arr.shape(0); ++i)
        sum += arr(i);
    return sum;
}

NB_MODULE(my_ext, m) {
    m.def("sum_array", &sum_array, "Sum a NumPy array (no copy)")
}
```

```python
import my_ext
import numpy as np
a = np.array([1, 2, 3], dtype=np.float32)
print(my_ext.sum_array(a))  # Output: 6.0
```

WHAT ARE TYPE CASTERS?

- **Type casters** are nanobind mechanisms that convert between Python and C++ types
- Built-in type casters handle fundamental types, STL containers, Eigen, and NumPy arrays
- You can write your own type casters for custom types to control how data is exchanged between Python and C++

EXAMPLE: WRITING A CUSTOM TYPE CASTER I

- Custom type casters let you control how a C++ type is converted to/from Python
- Useful for integrating third-party or legacy types

```cpp
#include <nanobind/nanobind.h>
namespace nb = nanobind;

// A simple C++ struct not directly supported by nanobind
struct MyStruct {
    int value;
};
```

EXAMPLE: WRITING A CUSTOM TYPE CASTER II

```cpp
// Custom type caster for MyStruct
namespace nanobind {
template <> struct type_caster<MyStruct> {
    NB_TYPE_CASTER(MyStruct, const_name("MyStruct"));

    // Python -> C++
    bool from_python(handle src, uint8_t) {
        if (!nb::isinstance<nb::int_>(src))
            return false;
        value.value = nb::cast<int>(src);
        return true;
    }

    // C++ -> Python
    static handle from_cpp(MyStruct src, rv_policy, handle) {
        return nb::int_(src.value).release();
    }
};
}
```

OTHER POSSIBILITIES TO EXCHANGE DATA

Nanobind offers Bindings

```cpp
#include <nanobind/stl/bind_vector.h>

using IntVector = std::vector<int>;
IntVector double_it(const IntVector &in) { /* .. omitted .. */ }

namespace nb = nanobind;
NB_MODULE(my_ext, m) {
    nb::bind_vector<IntVector>(m, "IntVector");
    m.def("double_it", &double_it);
}
```

```python
>>> import my_ext
>>> my_ext.double_it([1, 2, 3])
my_ext.IntVector([2, 4, 6])
```

EXAMPLE: BINDING A SIMPLE CLASS

```cpp
#include <nanobind/nanobind.h>
namespace nb = nanobind;

class Point {
public:
  Point(float x, float y) : x(x), y(y) {}
  float norm() const { return std::sqrt(x*x + y*y); }
  float x, y;
};

NB_MODULE(my_ext, m) {
  nb::class_<Point>(m, "Point")
      .def(nb::init<float, float>())
      .def("norm", &Point::norm)
      .def_rw("x", &Point::x)
      .def_rw("y", &Point::y);
}
```

```python
import my_ext
p = my_ext.Point(3, 4)
print(p.norm())  # Output: 5.0
print(p.x, p.y)  # Output: 3.0 4.0
```

`class_` BINDINGS OVERVIEW

Feature	nanobind Method(s)	Description
Constructor	`.def(nb::init<...>())`	Bind C++ constructors
Method	`.def("name", &Class::method)`	Bind instance methods
Field (ro/rw)	`.def_ro("field", &Class::field)` `.def_rw("field", &Class::field)`	Bind read-only/read-write fields
Property (ro/rw)	`.def_prop_ro("prop", &Class::getter)` `.def_prop_rw("prop", &Class::getter, &Class::setter)`	Bind properties
Static Method	`.def_static("name", &Class::method)`	Bind static methods
Static Field	`.def_ro_static("field", &Class::field)` `.def_rw_static("field", &Class::field)`	Bind static fields
Static Property	`.def_prop_ro_static("prop", &Class::getter)` `.def_prop_rw_static("prop", &Class::getter, &Class::setter)`	Bind static properties

APACHE ARROW

- **pyarrow** implements the Apache Arrow columnar memory format: Array, ChunkedArray, Table, RecordBatch, etc.
- pyarrow is the **default backend** for Polars, an option in Pandas, and will be required in Pandas 3.0
- type casters for Arrow types are available as nanobind extension library

EXAMPLE: USING ARROW ARRAYS

```
#include <memory>
#include <nanobind/nanobind.h>
#include <nanobind_pyarrow/pyarrow_import.h>
#include <nanobind_pyarrow/array_primitive.h>
#include <arrow/compute/api.h>  // Include Arrow compute functions

namespace nb = nanobind;

NB_MODULE(test_pyarrow_ext, m) {
    static nb::detail::pyarrow::ImportPyarrow module;
    m.def("my_pyarrow_function", [](std::shared_ptr<arrow::DoubleArray> arr) {
        auto result = arrow::compute::CallFunction("multiply", {arr, arr});
        if (!result.ok()) throw std::runtime_error(result.status().ToString());
        return std::static_pointer_cast<arrow::DoubleArray>(result.ValueOrDie().make_arr
    });
}
```

BUILDING NANOBIND EXTENSIONS

- **CMake** is the recommended build system for nanobind extensions
- **sckit-build-core** is a build backend for Python that uses CMake to build extension modules

DISTRIBUTING EXTENSIONS: WHEEL BUILDING

- **Wheels** (`.whl` files) are the standard for distributing Python binary packages
- Wheels allow users to install pre-built native extensions without a compiler
- **Problem:** Compiling a shared object (`.so`) file on one Linux distribution/version may **not** work on another due to differences in the system C library (**glibc**) and other dependencies.

MANYLINUX

- For Linux, the **manylinux** standard ensures compatibility across most distributions by building against an old, widely-supported glibc version inside a special Docker image
- manylinux_x_y tagged wheels shall work on any linux distribution based on glibc version x.y or newer

WHAT ABOUT OTHER DEPENDENCIES?

- Many Python extensions depend on shared libraries (e.g., Arrow, OpenBLAS, custom C++ libs)
- These dependencies may not be present on the target system, or may have incompatible versions
- **Bundling** these libraries inside your wheel ensures your extension works everywhere [auditwheel]

BUILDING WHEELS

- Tools like cibuildwheel automate building wheels for all platforms (Linux, macOS, Windows)

```
# Example: Build a manylinux wheel using cibuildwheel
pip install cibuildwheel
pytohn -m cibuildwheel
```

- Configuration can be done via a `pyproject.toml` file
- The resulting `.whl` files can be uploaded to PyPI for easy installation via `pip`

LINKS

The documentation of nanobind is available at

An example project is available on GitHub which demonstrates:

- How to use nanobind with CMake and scikit-build-core
- How to use nanobind with Apache Arrow
- How to integrate `spdlog` for logging
- How to create wheels using `cibuildwheel`